02-分离关注点：软件设计至关重要的第一步

你好！我是郑晔。

上一讲我们讲了软件开发就是在解决问题。那问题一般是如何解决的呢？最常见的解决问题思路是**分而治之**，也就是说，我们要先把问题拆分开。在每个问题都得到解决之后，再把这些解决好的子问题以恰当的方式组装起来。如何分解与组合，就是我们要在软件设计中考虑的问题。

然而，在软件设计这个环节中，大部分人都把焦点放在了如何组合上，却忽略了至关重要的第一步：分解。你可能会觉得：“分解？我会啊，不就是把一个大系统拆成若干个子系统，再把子系统再拆成若干个模块，一层一层拆下去嘛。”

然而，在我看来，这种程度的分解远远不够，因为分解出来的粒度太大了。**粒度太大会造成什么影响呢？这会导致我们把不同的东西混淆在一起**，为日后埋下许多隐患。

为什么这么说呢？我来给你举个例子。

一个失败的分解案例

我曾经见过一个故障频出的清结算系统，它的主要职责是执行清结算。一开始我觉得，清结算系统是一个业务规则比较多的系统，偶尔出点故障，也是情有可原。

但是在分析了这个系统的故障报告后，我们发现这个系统设计得极其复杂。其中有一处是这样的：上游系统以推送的方式向这个系统发消息。在原本的实现中，开发人员发现这个过程可能会丢消息，于是，他们设计了一个补偿机制。

因为推送过来的数据是之前由这个系统发出去的，它本身有这些数据的初始信息，于是，开发人员就在数据库里增加了一个状态，记录消息返回的情况。一旦发现丢消息了，这个系统就会访问上游系统的接口，将丢失的数据请求回来。

正是这个补偿机制的设计，带来了一系列的后续问题。比如，当系统业务量增加的时候，数据库访问的压力本身就很大，但在这种场景下，丢数据的概率也增加了，用于补偿的线程也会频繁访问数据库，因为它要找出丢失的数据，还要把请求回来的数据写回到数据库里。

也就是说，一旦业务量上升，本来就已经很吃力的系统，它的负担就更重了，系统出现卡顿也就在所难免了。

这个补偿机制的设计是有问题的，问题的点在于，上游系统向下游推送消息，这应该是一个通信层面的问题。而在原有的设计中，因为那个状态的添加，这个问题被带到了业务层面。

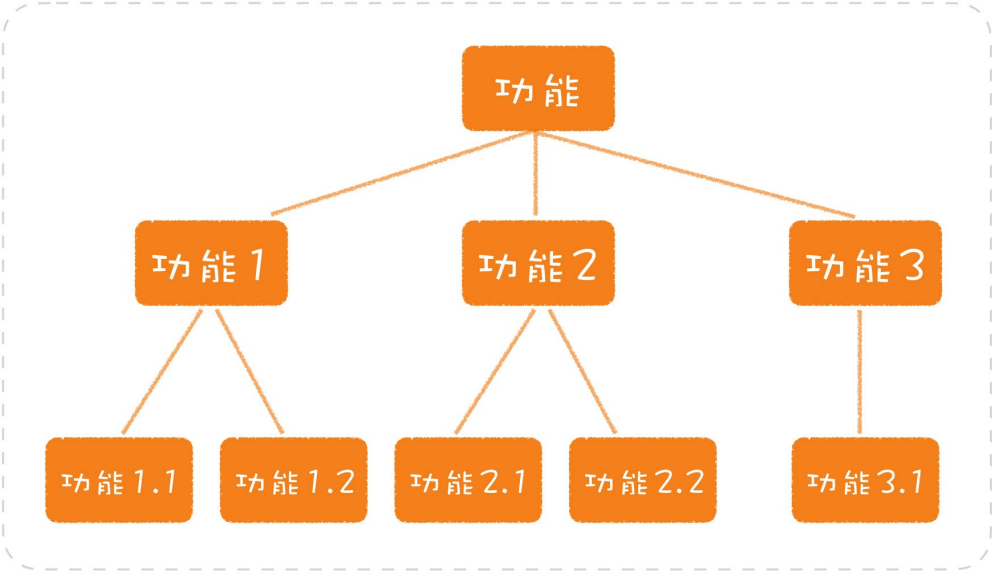
这就是一个典型的分解没有做好的例子，是分解粒度太大造成的。开发人员只考虑了业务功能，忽视其他维度。**技术和业务被混在了一起，随之而来的就是无尽的后患。**

一旦理解了这一点，我们就可以想办法解决了。既然是否丢消息是通信层面的事，我们就争取在通信层面解决它。我们当时的解决方案是，选择了一个吞吐量更大的消息队列。在未来可见的业务量下，消息都不会丢。**通信层面的问题在通信层面解决了，业务层面也就不会受到影响了**。果不其然，这样改造之后，系统的稳定性得到了大幅度的提升。

上面我只讲了这个故事的主线，其实，相关的事情还有一些。比如，上游系统专门为补偿而开发的接口，现在也不需要了，于是上游系统得到了简化；这个系统里那个表示状态的字段，其实还被用在了业务处理中，也引发过其他问题，现在它只用在业务处理中，角色单一了，与此相关的问题也少了。

分离关注点

至此，我们已经对分解粒度太大所造成的影响，有了一个初步的了解。那在做设计时，该如何考虑分解呢？传统上，我们习惯的分解问题的方式是树型的。比如，按功能分解，可分为：功能1、功能2、功能3，等等，然后，每个功能再分成功能1.1、功能1.2、功能2.1、功能3.1等等，以此类推。



如果只从业务上看，这似乎没什么问题。但我们要实现一个真实的系统，就不仅仅要考虑功能性的需求，还要考虑非功能性的需求。比如，前面提到的数据不能丢失、有的系统还要求处理速度要快，等等。

这与业务并不是一个维度的事情，我们在做设计时，要能够发现这些非功能性的需求。也就是说，我们在分解问题的时候，会有很多维度，每一个维度都代表着一个关注点，这就是设计中一个常见的说法，“**分离关注点（Separation of concerns）**”。

可以分离的关注点有非常多，你只要稍微注意一下，就能识别出来。但还有一些你可能注意不到，结果导致了混淆。最常见的一类问题就是**把业务处理和技术实现两个关注点混在了一起**，前面举的那个例子就是一个典型。

对于“把业务处理和技术实现混在一起”的问题，我再给你举个例子。如果现在业务的处理性能跟不上，你有什么办法解决吗？大多数程序员的第一反应是，多线程啊！

没错，多线程的确是一种解决办法。但如果不加限制地让人去把这段代码改成多线程的，一些多线程相关的问题也会随之而来。比如，让人头疼的资源竞争、数据同步等等。

写好业务规则和正确地处理多线程，这是两个不同的关注点。如果我们把二者放到同一段代码里去写，彼此影响也就在所难免了。问题说明白了，解决方案才能清楚，那就是把业务处理和多线程处理的代码分开。

按照我的理解，**大部分程序员都不应该编写多线程程序**。由专门的程序员把并发处理的部分封装成框架，大

家在里面写业务代码就好了。

把业务处理和技术实现混在一起，类似问题还有很多。比如我们经常问怎么处理分布式事务，怎么做分库分表等。其实，你更应该问的是，我的业务需要分布式事务吗？我是不是业务划分没有做清楚，才造成了数据库的压力？

在真实项目中，程序员最常犯的错误就是认为所有问题都是技术问题，总是试图用技术解决所有问题。**任何试图用技术去解决其他关注点的问题，只能是陷入焦油坑之中，越挣扎，陷得越深。**

另外一个常见的容易产生混淆的关注点是**不同的数据变动方向**。

有人问过我这样一个问题：在Java应用里，做数据库访问用Spring Data JPA好，还是MyBatis好。Spring Data JPA简化了数据库访问，自动生成对应的SQL语句，而MyBatis则要自己手写SQL。

普通的增删改查用Spring Data JPA非常省事，但对于一些复杂场景，他会担心自动生成SQL的性能有问题，还是手写SQL优化来得直接。是不是挺纠结的？

随即我又问了他一个问题，为什么需要复杂查询呢？他告诉我，有一些统计报表需要。

不知道你是否发现了其中混淆关注点的地方？普通的增删改查需要经常改动数据库，而复杂查询的使用频率其实是很低的。

从本质上说，之所以出现工具选择的困难，是因为他把两种数据使用频率不同的场景混在一起所造成的。如果将前台访问（处理增删改查）和后台访问（统计报表）分开，纠结也就不复存在了。

不同的数据变动方向还有很多，比如：

- 动静分离，就是把变和不变的内容分开；
- 读写分离，就是把读和写分开；
- 前面提到的高频和低频，也可以分解开；
- ……

不同的数据变动方向，就是一个潜在的、可以分离的关注点。

在实际的项目中，可以分离的关注点远不止这些。做设计时，你需要一直有一根弦去发现不同的关注点。分离关注点，不只适用于宏观的层面。

在微观的代码层面，你用同样的思维方式，也可以帮助你识别出一些混在一起的代码。比如，很多程序员很喜欢写setter，但你真的有那么多要改变的东西吗？实际上可能就是封装没做好而已。

分离关注点之所以重要，有两方面原因。一方面，不同的关注点混在一起会带来一系列的问题，正如前面提到的各种问题；另一方面，当分解得足够细小，你就会发现不同模块的共性，才有机会把同样的信息聚合在一起。这会为软件设计的后续过程，也就是组合，做好准备。

总结时刻

今天，我们学习了软件设计中至关重要的第一步：分解。

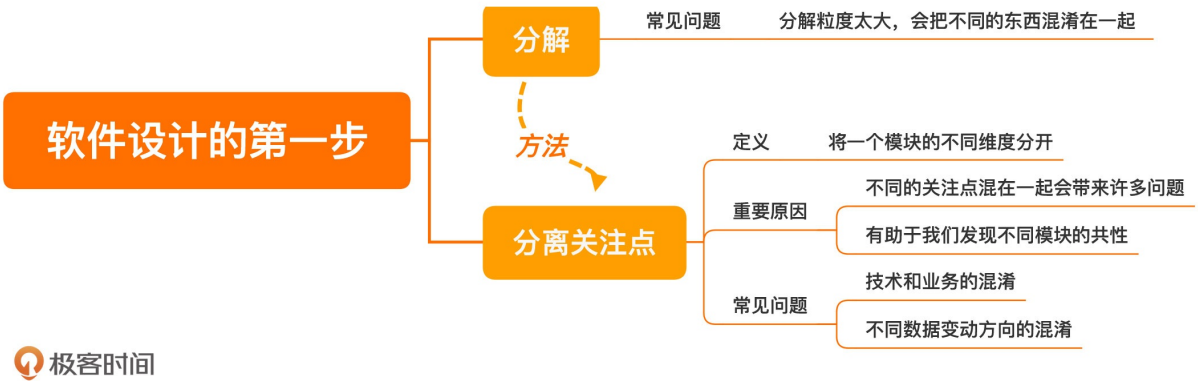
大多数系统的设计做得不够好，问题常常出现在分解这步就没做好。常见的分解问题就是分解的粒度太大，把各种维度混淆在一起。在设计中，将一个模块的不同维度分开，有一个专门的说法，叫分离关注点。

分离关注点很重要，一方面，不同的关注点混在一起会带来许多问题；另一方面，分离关注点有助于我们发现不同模块的共性，更好地进行设计。分离关注点，是我们在做设计的时候，需要时时绷起的一根弦。

今天，我还给你举了两种常见的关注点混淆的情况。一种是技术和业务的混淆，另一种是不同数据变动方向的混淆。希望你在日常开发中，引以为戒。

好，我们已经迈出了软件设计的第一步。接下来，就该考虑如何组合了。在组合的过程中，会有很多因素影响到组合的方式。下一讲我们就来看一个非常重要却不受重视的因素：可测试性。

如果今天的内容你只能记住一件事，那请记住：**分离关注点，发现的关注点越多越好，粒度越小越好**。



思考题

最后我想请你去了解一下CQRS（Command Query Responsibility Segregation），看看它分离了哪些关注点，以及在什么样的场景下使用这种架构是合理的。欢迎在留言区写下你的想法。

感谢阅读，如果你觉得这一讲的内容对你有帮助的话，也欢迎把它分享给你的朋友。

精选留言：

- 桃源小盼 2020-05-27 10:09:29
  能提供关于分离关注点更多的例子或者相关资料吗？ [7赞]

  作者回复2020-05-28 07:37:19
  专栏后面还会多次提到分离关注点的，敬请期待！

- Jxin 2020-05-27 10:52:52
  1.cqrs，命令与查询分离，最早是在ddd实战里面看到。其分离啦增删改与查询这两个关注点。

  2.静态上，拆分了这两块的代码。使各自可以采用不同的技术栈，做针对性的调优。动态上，切分了流量，能够更灵活的做资源分配。

  3.查询服务的实现。可以走从库，这有利于降低主库压力，也可以做到水平扩展。但需要注意数据延迟的

问题。在异步同步和同步多写上要做好权衡。
也可以都走主库，这时候查询服务最好能增加缓存层，以降低主库压力，而增删改服务要做好缓存的级联操作，以保证缓存的时效性。
当然也可以走非关系型数据库，搜索引擎类的es,solr，分布式存储的tidb等等，按需选择。 [6赞]

作者回复2020-05-27 21:17:37
非常棒的分享！

- 北天魔狼 2020-05-27 07:20:46
想起Kent Beck 说的一句话，大致意思是：我不准备在这本书里讲高并发问题，我的做法是把高并发问题从我的程序里移出去 [6赞]

作者回复2020-05-27 19:01:46
没错，就是这样。

- 飞翔 2020-05-28 07:21:30
老师 比如说订单系统 先下单写到数据库 然后发送消息给消息队列 这两部 没法放到一个事务中去。 如果用本地消息表， order 写数据库 然后 在写本地消息表 这样这两步就放到一个事务中去了 保证肯定成功，然后在有线程 读取本地消息表 发送队列 如果成功更改本地消息表状态 。 从设计角度讲这就没分离关注点， 这个应该怎么分呀？ [4赞]

作者回复2020-05-28 11:12:46
我们来分析一下这个需求，下单入库和发消息给下游，这确实是两个动作，但这两个动作的顺序一定是这样吗？它们一定要在一个线程里完成吗？

我们可不可以先发消息呢？比如，我们把消息发给下游之后，有一个下游接收到消息之后，再把消息入库。如果这样做的话，发消息，由消息队列保证消息不丢，下游入库，又可以保证订单持久化。你看，在这个设计中，其实，并不需要事务，所以，我们也不必为事务纠结了。

- 我是小妖怪[C|N] 2020-05-27 10:37:09
有感觉，但是又不明确，没有get到那个点，应该举一下具体的业务来说明或者证明，感觉是理论上的 [4赞]

作者回复2020-05-28 18:24:18
你把你困惑的点提出来，我争取进一步讲清楚。

我在部落里写了一个回答，可以参考一下。
http://gk.link/a/10iHp

- Kǎfkā^{2o2o} 2020-05-27 23:28:14
近期有一本书《被统治的艺术》，正好和软件设计中的职责分离策略异曲同工。

我们知道明朝自朱元璋开始有一个顶层设计，就是每家每户做什么，一开始就规定好了。军队也是一个固定职业，即军户制。比如说国家需要100万个士兵，那就要有100万个军户，每户出一个兵，世世代代都是这样。如果这个兵逃了或者死了怎么办？家族里就再出一个来补充。

这会带来什么后果呢？你可以想象一下，如果儿童节的时候你正坐在家里跟妻子儿女享天伦之乐，忽然有人闯进来，把你抓走了去当兵，只是因为你家族里面的另外一个人当了逃兵或者死掉了。

可见，这样的顶层设计会给自己的家族带来各种不确定性甚至家庭悲剧。人民群众想出了很多的策略来对付这样的制度。

有种设计是这样的，就是每个家族中选出一个分支代表整个家族去当兵，与之相对的是家族的其他分支需要共同出一笔钱，世世代代赡养这个当兵的分支。此外还有其他一些「福利」，比如说，如果原本他在家族中的排位比较低，那他的后代就可以在家族的各项活动中提升座次。

这个世世代代当兵的分支会比较惨，但带来的好处是这个家族中的其他分支就会少受骚扰，得以繁衍。

事实上这样的策略运行得不错，有些家族好几代人一直都执行这样的策略，甚至贯穿了几乎整个明代。

某种角度说，这就是一种职责分离，将国家统治的要求和家族稳定繁衍的需要分开。 [3赞]

作者回复2020-05-28 07:17:55
刚好最近万维钢老师讲了这本书中的内容，但你从软件设计的角度去理解这个问题，确实让人有一种耳目一新的感觉。

- 业余爱好者 2020-05-27 00:22:41
技术和业务混杂的情况，让我想起来一篇文章，大意是说要区分技术异常和业务异常的。也就是说，技术层面的异常信息不应该暴露给上层的业务人员。典型的例子就是大型网站的错误页面，而不是直接把后台的npe堆栈信息抛给用户。 [3赞]

作者回复2020-05-27 19:02:59
这是一个很好的例子，确实要做区分。

- 飞翔 2020-05-27 22:09:00
老师 能具体说说加了消息队列的数据流成什么样了 为啥能解决对消息问题呀 [2赞]

作者回复2020-05-28 06:31:47
这里并不是说增加了消息队列解决的问题，原有的解决方案用的也是消息队列。

这里的重点是，用了一个吞吐能力更强的队列，保证了消息的不丢失，这样我们就不必专门处理消息丢失的问题了。通信的问题在通信的层面得到了解决，就不会影响到其它的代码了。

- 夏天 2020-05-27 13:21:27
我发现大家在工作中往往不做分离，分析需求的时候把方案揉在一起。
可以怎样去练习做分离呢？ [2赞]

作者回复2020-05-27 20:35:01
有一种从小事练起的方法，就是写代码时，把自己写的函数行数限定在一定的规模之下，比如，10行。超过10行的代码，你就要去仔细想想是否是有东西混在了一起。

这种方法锻炼的就是找出不同关注点的思维习惯，一旦你具备了这种思维习惯，再去看大的设计，自然也会发现不同的关注点。

- 光明 2020-05-27 09:47:38
1. 在软件设计中，大家是期望将粒度分解的越小越好，但又往往嫌分解太小过于麻烦。就像，希望别人把文档写好，自己却又不写(￣_,￣)
2. 业务处理和技术实现很容易被混在一起，原因也确实是分离的不够┬┴┬┴﹏┬┴┬┴
[2赞]

作者回复2020-05-27 21:18:23
太过真实了：想好，又不希望自己做得太多。

- Geek_3b1096 2020-05-28 11:42:59
  一身冷汗，要从根本上提升分解能力 [1赞]

  作者回复2020-05-28 18:37:56
  赶紧分解起来。

- 阳仔 2020-05-28 09:53:50
  注意分解粒度，分离关注点，这些很重要，
  但问题是如何分解，如何分离关注点，具体有哪些工程实践？ [1赞]

  作者回复2020-05-28 18:21:24
  我在部落里面写了一个回答，可以看一下。

  http://gk.link/a/10iHp

- 阳仔 2020-05-28 09:41:31
  软件设计第一步是对业务功能进行分解，如何分解是有讲究的，分解子功能的粒度尽量要小。文中提到交易原语的例子，通过子功能来实现大功能的逻辑，其实也就是分层的思想。每一层关注自己的业务逻辑，对外提供接口 [1赞]

- 算不出流源 2020-05-27 23:21:41
  “如果将前台访问（处理增删改查）和后台访问（统计报表）分开，纠结也就不复存在了。”
  老师请恕我愚钝，所以将高低频分开之后是分别采用Spring Data JPA和Mybatis来实现进行数据库访问吗？如果是的话，那不是相当于在同一个项目中引入了两套数据库访问规范，会不会造成开发规范上的困惑甚至混乱？如果不是的话，那正确做法又应该是什么？ [1赞]

  作者回复2020-05-28 06:35:11
  如果把二者分开，这可以就是两个项目，一个前台项目，一个后台项目。两个独立的项目各自采用一套编程规范，不就很正常了。

- gsz 2020-05-27 14:02:39
  看完貌似懂了，细想完全没懂 [1赞]

  作者回复2020-05-27 21:16:50
  欢迎把困惑提出来，我争取帮你解惑。

- olym 2020-05-27 12:59:42
  分解的粒度一般到什么样的层级才能更好的分析共通性以及更好的组合呢？ [1赞]

  作者回复2020-05-27 20:36:33
  这就是我建议的由来，越小越好。最小的粒度就是函数，函数写得越小越好。怎么写小呢？就是要分解出更小的粒度，这是一种练习的方法。

- 小豹哥 2020-05-27 10:03:55
  经典 [1赞]

- 北天魔狼 2020-05-27 07:16:20
  API功能易变，开发原则易扩展；
  后台展示管理，基本就是看，用的框架生成的；
  老师，这个算分离关注点吗？ [1赞]

作者回复2020-05-27 21:18:59
这个理解总体上来说，是没错的。

- 捞鱼的搬砖奇 2020-05-27 02:11:55
想到10x的任务分解 [1赞]

作者回复2020-05-27 19:02:20
任务分解和分离关注点确实是异曲同工。

- Kăfkă²º²º 2020-05-29 07:48:19
最近听到大家的一些技术设计。有些同事在偏业务数据驱动的系统里用了juc里的比如原子操作。虽然对大家提升技术水平有一定益处，但从软件设计角度来说，未必是好事，应该尽力避免。